P0P Design-Test Document

Partner A:Matthew Jagen

Partner B: Koshik Mahapatra

PART 1

1. Server Design

Overview

Our server runs using three threads. One for handling user input via the terminal, another for managing the socket, and a third to handle timeouts. The main thread is the IO thread so that the other threads can be daemon threads and terminate when the IO thread terminates from EOF. The main loop in the IO thread is a for loop for each line in sys.stdin so that it blocks until there is input from the terminal. The socket thread waits for a new packet in the socket every iteration and then evaluates the packet and responds accordingly. This thread also handles all of the print messages. For the timer thread, our implementation checks to see if any of the active sessions had a packet time out each time it wakes up. If one timed out, it sends GOODBYE to the client and closes the session. After handling any timeouts, the timer thread calculates the next time a session might time out and calls time.sleep() until then.

Justification

We tested our server's efficiency against the barebones implementation from part one. For both the complete and barebones server we had one client send the Dostoyevsky.txt file without waiting for the server's response after each packet. In the case where the servers printed a "Lost packet!" message for each packet that was lost, the barebones client handled an average of 2,103 packets each time, and the fully implemented server handled an average of 2,225 packets each time (we ran each test 3 times). When we removed the loop that printed out the "Lost packet!" message, the barebones server handled an average of 9,613 packets each time, and the fully implemented server handled an average of 8,952 packets each time. These numbers are similar enough that we are confident in our full implementation of the server. For multiple clients, our design kept a list of every active session which meant that it came with the added inefficiency of iterating through the list to find the session's data for each packet. When the Dostoyevsky.txt file was sent to the server from two clients, the server received an average of 3,941 packets each time, and when the file was sent from three clients, the server received an average of 3,880 packets. Since all of the clients used the same socket, it is expected that the server would lose more packets the more clients there are, but the small difference between two and three clients shows our server can handle multiple clients well.

Data structures

ClientData class

The ClientData class tracks all of the information associated with a single session. It tracks IP, port, session ID, sequence number, and the next time a timeout could occur. As for synchronization, the IO thread only reads ClientData objects when the server is shutting down and the other threads are not using them. The timer thread only reads the ClientData object's timeout variable and it doesn't matter when the socket thread updates the timeout value because the timer thread will just notice it was changed the next time it wakes up.

List of ClientData objects

The data structure we use to track all of the sessions a server has open is the built-in python list() structure. We use this data structure because we only ever add to the back and python handles synchronization automatically for it.

How timeouts are handled

When the timeout thread finds out that a timeout has occurred, it calls the same close_session() function that the socket thread calls when it closes a connection. This function sends a GOODBYE message to the client, prints a session closed message and removes the ClientData object associated with the connection from the sessions list.

How shutdown is handled

When the IO thread receives 'q' or EOF as the user input, it closes all of the client connections by iterating through the sessions list and calling close_session() for each ClientData object. Then, once it has closed all of the sessions, it closes the socket and exits. Because the IO thread is our main thread and we made all of the other threads daemon threads, the others threads exit as well when the IO thread exits.

Any libraries used

The thread-based server implementation only uses the Python standard library. It uses the standard library for threading, sockets, and for encoding/decoding data packets received as Bytes objects.

Corner cases identified

Describe corner cases that you identified and how you handled them.

- Packet received is smaller than 12 bytes (minimum size for p0 packet header)
    - ignore the packet
- Packet received has invalid magic or version number
    - ignore the packet
- P0 packet header is a HELLO message with a non-zero sequence number
    - send GOODBYE

- Client sends ALIVE packet
    - send GOODBYE
- Packet received from different IP/port than was previously used for that session
    - ignore the packet
- Duplicate packet
    - ignore the packet
- Out of order packet (not including wrap-around sequence num)
    - send GOODBYE
- HELLO from previously existing session ID
    - send GOODBYE

2. Server Testing

Testing your server

To test our server, we used the python standard library's unittest framework to mimic different client behaviors. We have a total of 9 test cases that each test the server's response to varying edge cases. We tested all of these on the UTCS lab machines in a minimal bash environment. The first test case is a baseline test that sends HELLO, DATA, and GOODBYE packets in that order and checks the expected value of each response packet (including the magic and version numbers, the appropriate command, sequence number, and session id). The second test is a similar baseline test but only sends HELLO and GOODBYE. The third test tests the edge case where the client goes crazy and sends an ALIVE. The test then checks to make sure that the server recognized the protocol error and responded with GOODBYE. The fourth test case sends a DATA packet with no data and expects to be handled normally (server responds with ALIVE). The fifth test case sends a duplicate HELLO packet with sequence number 0 and expects it to be ignored. The sixth test case sends a second HELLO packet with a non-zero sequence number and expects the server to recognize it as a protocol error. The seventh test case sends a packet that is 2 bytes long and expects the server to ignore it. The eighth and ninth test cases send packets with bad magic and version numbers respectively and expect the server to ignore them. All of these test cases are in the ./Thread/test_client file.

3. Client Design

Overview

Thread Based

The thread based client has three threads. A thread that handles user input and sending that input to the server, a thread that handles the responses from the server, and a final thread that handles timeouts. The thread that listens to the server's response is the main thread, which was done so that when the client receives a GOODBYE message the main thread can easily close and kill off the other two daemon threads. The thread that listens to the server is primarily in a loop that blocks on the socket's recvfrom() method, and in each iteration it handles the freshly received packet. If the packet it received is a protocol error it returns from the

listen() function with a code of 1, signifying that the program should close immediately. If the packet was not a protocol error, it handles it appropriately and resets the timeout time that the timer thread checks. The thread that reads user input blocks on waiting for input from the terminal and in each iteration checks that the input isn't EOF/q and then exits or sends the data to the server. Lastly, the timer thread sits in a while True loop and continually sleeps until the next timeout time set by the socket thread. If the timer thread wakes to find out that the timeout time has passed, it alerts the socket thread to exit by sending a dummy packet.

Event Based

The event based client is structured very similarly to the thread-based client with three different events that it waits for. Our pyuv main loop has UDP, TTY, and Timer objects to handle all of the events. Upon receiving input from the terminal, the TTY invokes the callback method we titled "readstdin()" and that function ends the loop on EOF/q and otherwise converts the input into a DATA packet and sends it to the server. When the pyuv UDP object receives a packet from the server, it calls the receive() function which handles the packet and will either reset the timer by calling timer.again() or will handle GOOBYE/protocol error by closing. Lastly, the timer object calls timeout() when the timer goes off which will either send GOODBYE and transition into the closing state or, if already in the closing state, will end the loop.

Justification

Justify why your design above is efficient. Also show evidence how your client can handle sending packets from a large file (each line is close to UDP packet max and also contains many lines) and receiving packets from the server at the same time. Note you do not want to cause the false TIMEOUT from server because you are too busy just sending out the packets to the server when the server actually has sent you a packet before the TIMEOUT.

We know our client is efficient because we have tests that test sending from large files (Dostoyevsky), as well as tests that send lines close to the max UDP packet size. It can also handle receiving and sending packets at the same time without worrying about a timeout because the timeout thread will always wake up the next time a timeout could have happened.

Data structures

Both the event based and the thread based client do not use any data structures. All they use are global variables to track the sequence number/session ID as well as the current state.

How timeouts are handled

In the case of the thread based client, when the timer thread awakens to find out that a timeout has occurred, it sends a dummy packet to the socket to alert the main thread that a timeout has occurred. The main thread then transitions to the closing state and handles closing the program

How shutdown is handled

When the client receives EOF or q, it transitions to the closing state, where it sends a GOODBYE message to the server and waits for the server's response. The main thread blocks on the socket after setting another thread to sleep for 5 seconds. If the main thread receives a GOODBYE from the server in response, it simply exits and brings the other thread with it since it is a daemon thread. If the timer thread wakes up, it knows a timeout happened and sends a dummy packet to the main thread so that it unblocks and can exit.

Any libraries used

For the thread based client we only used the python standard library for threading and the socket. For the event based client we used pyuv to handle the event loop and all of the handles in the event loop (UDP, TTY, and Timer).

Corner cases identified

Describe corner cases that you identified and how you handled them.

- Packet received is smaller than 12 bytes (minimum size for p0 packet header)
    - send GOODBYE
- Packet received has invalid magic or version number
    - protocol error, close immediately
- P0 packet header is a HELLO message with a non-zero sequence number
    - protocol error, close immediately
- Packet received is duplicate HELLO
    - ignore packet
- Client receives DATA packet
    - protocol error, close immediately

4. Client Testing

Testing your client

We tested our client by making an Expect file that covers things that the naive.exp and Dostoevsky.exp files didn't cover. We tested it both on the same lab machine, and from one lab machine to another. The first set of test lines sends a large amount of empty lines to the client for it to send to the server. The second set of test lines sends 9 lines of input that are close to UDP-max length to the client for it to send to the server. These tests are in ./Event/testclient.exp

5. Reflection

Reflection on the process

What was most challenging in implementing the server? What was most challenging testing the server?

The greatest challenge in implementing the server was getting the threads to work together. More specifically, coding the timer and IO threads in a way that when there was a timeout/eof they would close the sessions. Testing the server was overall pretty easy. The hardest part for testing the server was that there were a lot of edge cases to check.

What was most challenging in implementing the client? What was most challenging testing the client?

The most challenging part of implementing the client was implementing multithreading in a way that allowed multiple different things to cause the client to exit. For example, in the closing state, the timer timing out or th
e socket thread receiving GOODBYE should both cause the client to exit, but it was difficult to find a way to get one thread to close another and vice versa. The hardest part for testing the client was that we couldn't structure our tests in the same way as the server (using python's unittest framework) since the client isn't persistent like the server is. Since we couldn't really test the client by emulating the server we instead opted to use Expect to emulate client input.